

Binning

1 Binning Conceptual Model

The proposed binning conceptual model is comprised of pass and fail soft bin groups, counters, device specs, zero or more soft-to-hardware bin maps, bin setting mechanisms, bin based events, and notions of a null bin and bin arbitration.

The motivation behind these elements and the structure imposed on them, besides the obvious classification of devices based on test results and collection of granular test data, is to enable writing easy to maintain code. Specifically, we exploit the relationship between device specifications and software bins so that editing (adding, deleting, enabling, disabling) device specifications and bins can alter the execution of a properly designed flow without having to edit the code describing that flow.

1.1 Top Level Container

This container has an id and contains general information and two groups of soft bins: pass bins and fail bins¹. The conceptual model for each group is the same hence, one syntax should suffice for both.

The general information consists of a user-settable/readable property *ContinueOnFail* which is used as the default on bins for which this property is not explicitly set.

1.2 Soft Bin Group

Each group contains one or more axes which in turn contain soft bins. If the group contains only one axis, the axis may be anonymous. If the group contains two or more axes², each axis must have an identifier unique to the group. Each axis contains one or more soft bin definitions. Each soft bin definition identifier must be unique to the axis.

ContactOpens	ContactShorts	Functional	Timing
5	5	6	7

Table 1: Fail bin group with anonymous axis

	3.00GHz	2.93GHz	2.66GHz	ClockSpeed
8Mb	1	3	3	
4Mb	2	4	4	
CacheSize				

¹ Teradyne has type *error* in addition to *pass* and *fail*. Handle errors via *OnError* entry point with no special bin category?

² Explore the use of nested axes.

Table 2: Pass bin group with two labeled axes

Table 1 and Table 2 are representations of typical soft bin groups. Axis labels are shown in bold white on black background and soft bin names are shown in bold black on white. Table 1 is in effect a 1 x 4 array, i.e., 1 anonymous axis with 4 soft bin definitions. If there is a soft-to-hard bin map, each of the array elements shown in yellow must be mapped to a hardware bin. Hardware bin numbers imposed by an associated bin map are shown in green.

Table 2 is a 3 x 2 array, axes labeled *ClockSpeed* and *CacheSize* which have 3 and 2 soft bins respectively. Again, if there is a soft-to-hard bin map, each of the array elements shown in yellow must be mapped to a hardware bin.

The group has user-settable/readable property *color* of type *String* which serves as the default color for bins under that group, and a read-only property which returns the number of Axes in the group as an *Unsigned*.

1.3 Soft Bins

1.3.1 Null Bin

The null bin is used to define generalized actions. For example, a TestBase fail action for most production tests or flows derived from it, is to bin and stop, i.e., a single action that takes a bin argument: if the argument is null, it neither sets the bin nor stops, if it is a user-defined bin it sets the bin and then stops³.

³ Set TestBase fail bin data member to null and override the fail bin data member of the derived test or flow with a user-defined bin. Define stop action semantics, e.g., trigger an entry point where software bin arbitration can be done? Does it unwind the stack and carry out post actions ?

1.3.2 User Defined Bins

Data Type	Data Description	Comment
Const String	color	Hexadecimal RGB value or text, e.g., red. Used for print or display purposes. Default value is the group color.
Boolean	enable	Disabled bins are treated as though they didn't exist at run-time. Bins are enabled by default.
Const String	general descriptor	May require several, e.g., character for a wafer map, short string for restricted real estate descriptor, and verbose descriptor.
Const enumerated/string/id	name	Allow for conversion to number or spec storage ? See <i>user defined name</i> in 1450.0.
Const Integer	number	Bin number is optional. Default value is the index (starts at zero).
Const Boolean	pass/fail	Whether a bin is a pass or a fail bin is determined by the group in which it is defined
Mutable Unsigned	retest	Non-zero causes retest on fail up to N times ⁴

Table 3: User-settable soft bin properties

Table 3 shows only the bin properties that are programmed directly by the user via definition syntax. User-defined bins may be referred to by name or number and interrogated for additional information (see section 1.8 and Table 4).

Data Type	Data Description	Comment
Const Unsigned	index	The index into the Axis, i.e., array of bins.
Const Unsigned	counter	The number of times a bin has been set. See section 1.6 and Figure 6
Const Boolean	is_set	True if the bin is set. See Figure 6.

Table 4: Read-only soft bin properties

⁴ If the program stops and retest is non-zero, retest is decremented and an OnRetest event is generated. If retest is zero on stop, the program stops and restores retest's original value.

1.4 Hard Bins

Same as user-defined soft bins (see Table 3 and Table 4) if required. Question: are hard bin definitions required or is an integer representing a hard bin sufficient⁵ ?

1.5 Bin Setting and Arbitration

Bins are automatically unset by the OnStart event handler. When a bin is set, it remains set until the next OnStart event. Bins may be set only in the context of FlowNode, Test, and/or Flow (post?) actions via one of the mechanisms described in Table 5.

Mechanism	Parameters	Action
Bin	1. bin name, number, or variable	Set bin and continue.
SetBinStop ⁶	1. bin name, number, or variable 2. Boolean ContinueOnFail	Continue if parameter 1 is the null bin, otherwise set bin and based on parameter 2 either stop (generates OnFinish event) or continue.
SetBinRetest	1. bin name, number, or variable 2. Boolean ContinueOnFail 3. Unsigned Integer retest	Same as SetBinStop with additional retest actions. Retest schemes: <ol style="list-style-type: none"> If the program stops and retest is non-zero, retest is decremented and an OnRetest event is generated. If retest is zero on stop, the program stops and restores retest's original value. make retest Boolean and use an unsigned Integer data element (in the softbin group container?) to control the maximum number of reprobes. provide bin independent retest syntax.

Table 5: Soft Bin Setting Mechanisms

Looking back on the Table 2 example, independent tests may reach different conclusions regarding device speed, e.g., an at-speed functional test passes the device at 2.93GHz whereas a timing test determines that the device meets its 3.00GHz spec. For various

⁵ That integer is in the Map statement in the BinMap block.

⁶ Stop, a related mechanism, has nothing to do with binning.

purposes, hardware binning being among them, one of the two bins, presumably the lesser 2.93GHz bin is chosen to act upon.

An Axis is an object that returns, e.g., the highest index bin that is set when queried.

If, in the Table 2 example, the at-speed functional test fails and a timing test still determines that the device meets its 3.00GHz spec, then the fail bin should be acted upon.

If multiple fail bins are set the lowest (highest) index bin is acted upon.

Alternatively or additionally, the syntax can provide means for interpreting any combination of pass and fail bins that have been set.

1.6 Counters

1.6.1 Soft Bin Counters

Proposed: there is a counter group for each object in the binning hierarchy beginning with the top level. Each counter in the group is initialized/re-initialized by a different event (see Table 6 for a list of supported events) and automatically incremented when a bin is set⁷. The counters are chained such that incrementing a group at the lowest level, increments every group above it. The entire structure is repeated for each site.

1	OnLoad
2	OnLotStart
3	OnWaferStart
4	OnRetest
5	OnStart

Table 6: Counter group consists of one counter for each of these events

The Table 1 / Table 2 example is used to describe the counters mechanism. Assuming single site, there are 75 counters in this example, one group per each of the fifteen objects in the hierarchy shown in Table 7 times five counters per group, one counter per event shown in Table 6.

⁷ The language provides read-only access to these counters.

As the events in Table 6 occur, each of the corresponding counters in the 15 groups of Table 7 is initialized.

1	Top Level	Soft Bin Groups	Axes	Bins
2		Pass		
3			ClockSpeed	
4				3.00GHz
5				2.93GHz
6				2.66GHz
7			CacheSize	
8				8Mb
9				4Mb
10		Fail		
11			Anonymous	
12				ContactOpens
13				ContactShorts
14				Functional
15				Timing

Table 7: a counter group per row per site

When a bin is set, the corresponding counter hierarchy is incremented. For example, a timing test sets the 2.93GHz bin, incrementing the 2.93GHz counter group (Table 7, row 5), the ClockSpeed counter group (Table 7, row 3), the Pass counter group (Table 7, row 2), and the Top Level counter group (Table 7, row 1).

Support bin yield alarms (too high, too low).

1.6.2 Hard Bin Counters

Same as soft bin counters, i.e., is there any reason for the conceptual model to be different ?

1.6.3 Counter Based Events

These are events triggered by counter contents or calculations involving counter contents and handled by *EntryPoints*, e.g., scrubbing and/or re-probing for up to three consecutive

contact test failures, characterizing every Nth good device, or stopping and raising an alarm when device yield drops below a certain point..

1.7 User-defined Device Specifications

Proposed: for software maintenance purposes it is useful to allow one or more sets of user-defined device specs per axis. The intent is to be able to write tests that can iterate simultaneously over axis bins and their corresponding specs. To be used for this purpose, each named specification must be able to hold a one or more values (min, max, or min and max) per bin⁸.

In our example, one set of device specifications may be used for ac timing tests for a micro-processor. A single timing test can then iterate over each spec, e.g., data setup time, iterate over the timing values associated with the 3.00GHz, 2.93GHz, and 2.66GHz portions of that spec and bin the device accordingly.

Adding another spec, e.g., data hold time, then becomes a matter of adding in effect, a named variable with three values (assuming minimums only), one for each bin under *ClockSpeed*. Adding another *ClockSpeed* becomes a matter of adding a bin and a corresponding value to each of the specs associated with the *ClockSpeed* axis. Disabling one or more *ClockSpeed* bins can be implemented as a simple Boolean function.

1.8 Data Access

Proposed: access is provided for both stored and processed data, e.g., the contents of a counter (stored) and the dominant bin if more than one is set⁹ (processed).

Stored data may be accessed via the object hierarchy using, e.g., a dot syntax: *a.b.c* where data element *c* is contained by object *b* which is contained by object *a*. If *b* is an array then *c* of the first element of *b* may be accessed by, e.g., *a.b[0].c* (consider associative arrays for binning, i.e., the index may be other than numeric).

Processed data may be accessed via the object hierarchy using, e.g., a dot syntax: *a.b.c()* where function *c* is a member of object *b* which is contained by object *a*.

Appropriate member functions need to be defined for each object type.

2 Soft to Hard Bin Maps

Proposed: we provide the capability to define zero or more named maps and a mechanism for selecting the currently active map. Each map has no less than one entry for each cell in the array of enabled software bins formed by the bin axes, and no more than one entry

⁸ STIL.0 *Spec* and *Category* syntax may be applicable here: *Spec* per *Axis*, *Category* per *Bin*.

⁹ The dominant bin is used for soft to hard bin mapping, for example.

for each cell in the array of enabled and disabled software bins¹⁰. For single axis bin definitions, that means one entry per software bin. For multi-axis arrays, that means one entry for each combination of software bins. Each entry maps a software bin array cell to a hardware bin, specified as an unsigned Integer.

Assuming all bins are enabled, the example used in this document requires one entry for each fail-bin in the anonymous one dimensional array, and one entry for each of the combinations of ClockSpeed/CacheSize axis bins, i.e., 3 x 2 equals 6 entries.

3 Binning Syntax

3.1 Definitions

```
softbin_property =  
<  
    Color = String;           |           // Hex RGB or name  
    Enable = Boolean;         |  
    Number = Integer;        |           // 2 bins with same number OK  
    Retest = Unsigned;       |           // Hold off on semantics  
    Terse = String;          |  
    Verbose = String;        |  
    WafermapChar = Character;  
>  
softbin_definition =  
    Bin SOFTBIN_NAME; |           // Sets property Name  
    Bin SOFTBIN_NAME { softbin_property* }
```

Figure 1: Soft bin definition

¹⁰ In other words, disabled bin cell entries are optional.

```

BinDefs BINDEFS_NAME {
  (ContinueOnFail = Boolean;) // Contained bin default, false if unspecified
  Pass {
    (Color = String;) // Contained bin default, "green" if unspecified
    (softbin_definition* |
     ( Axis AXIS_NAME {
       softbin_definition*
     } )*)
  }

  Fail {
    (Color = String;) // Contained bin default, "red" if unspecified
    (softbin_definition* |
     ( Axis AXIS_NAME {
       softbin_definition*
     } )*)
  }
}

```

Figure 2: Soft bin definition block

```

BinMap BIN_MAP_NAME {
  ( Map (Pass.|Fail.)(AXIS_NAME.)SOFTBIN_NAME Integer; )* |
  ( Map [ (Pass.|Fail.)(AXIS_NAME.)SOFTBIN_NAME)+ ] Integer;
)*

```

Figure 3: Soft to hard bin mapping block

Bring Figure 3 in line with chosen example which pinpoints a cell by specifying row and column (allow enumerated type instead of Integer?):

```

Map [ Pass.ClockSpeed."3.00GHz" ] [Pass.CacheSize."8Mb" ] 1;
Map [ Pass.Axes[ClockSpeed].Bins["3.00GHz" ] ] [Pass.
Axes[CacheSize].Bins["8Mb" ] ] 1;

```

or, since the two bin descriptors are unique in the context of the BinDefs block being mapped:

```

// Allow optional Axes, Bins?
Map Pass[ClockSpeed]["3.00GHz" ] Pass[CacheSize]["8Mb" ] 1;
Map Pass["3.00GHz" ]["8Mb" ] 1;

```

The bracketed Map statement requires that all bin descriptors refer to the same group, i.e., either Pass or Fail, and that each bin listed comes from a different axis in that group. Each Map statement should contain a unique combination of bins. The totality of Map

statements should cover all combinations. For the example used in this document, that requires $3 \times 2 = 6$ Map statements to map all Pass bins..

Changes from P1450-4-D18-SyntaxSummary11-28-2007.pdf:

- Removed Integer option in “**Bin SOFTBIN_NAME (Integer);**” because:
 - in the context of *Axis* it is unclear what the relationship between that Integer and the index of a *Bin* is.

a gap in the number sequence and duplicate integers pose problems with iterating over *Bins* if that Integer is the index.

- Table 3 Added individual soft bin properties as per Table 3. This captures the intent of the Integer above using a different syntax.
- Added keyword Map to each line under *BinMap* to conform with 1450.0 general STIL statement form: **Keyword (OPTIONAL_TOKENS)*;**
- Required at least one soft bin in brackets for 2nd statement under *BinMap*.
- Substituted white-space for `->` in statements inside the *BinMap* to mimic 1450.1 *NameMaps* syntax (that syntax puts the use of keyword Map in question):

```
NameMaps VECTOR_ASSOCIATIONS {
    Signals {
        "A" "top_test.PI[0]";
        "B1" "top_test.PI[1]";
        "C1" "top_test.PI[2]";
        "D11" "top_test.PI[3]";
    }
    SignalGroups {
        _PI "top_test.PI";
        _PO "top_test.PO";
    }
    Variable { _PATCOUNT "PATTERN"; }
}
```

3.2 Usage and Data Access

3.2.1 Group Property Access Syntax

```
group_property =
<
    Color          | // String (hexadecimal RGB or name, e.g., red
    isAnyBinSet   | // Boolean
    Size          | // Unsigned, number of Axes
>
```

Figure 4: Group Property Access

3.2.2 Axis Property Access Syntax

```
axis_property =  
<  
  highestSetBin      | // Unsigned(Integer in case no bin is set ?)  
  lowestSetBin       | // Unsigned(Integer in case no bin is set ?)  
  Size               | // Unsigned, number of Bins  
  Name               | // String  
>  
group = < Pass|Fail >  
  
(BINDEFS_NAME.)group[Integer/AXIS_NAME].axis_property
```

Figure 5: Axis Property Access

3.2.3 Bin Property Access Syntax

```

counter_reset_event =
<
  OnLoad
  OnLotStart
  OnRetest // On Keyword, e.g., On Retest
  OnStart
  OnWaferStart
>
bin_property =
<
  Color // String
  ContinueOnFail // Boolean – Reconsider under Phase2
  counter.counter_reset_event // Unsigned
  Enabled // Boolean
  Index // Unsigned
  IsFailBin // Boolean
  isSet // Boolean
  Name // String
  Number // Integer
  retest.(current|Original) // Unsigned
  Terse // String
  Verbose // String
  WafermapChar // Character
>
group = < Pass|Fail >

(BINDEFS_NAME.) group.Bins[Unsigned|SOFTBIN_NAME].bin_property |
(BINDEFS_NAME.)
group.Axes[Unsigned|AXIS_NAME].Bins[Unsigned|SOFTBIN_NAME].bin_property

```

Figure 6: Bin Property Access

For example, the expression¹¹:

```
Pass.Axes[ClockSpeed].Bins["3.00GHz"].Index
```

yields 0, the index of bin "3.00GHz", whereas:

```
bindefs.Pass.Axes[ClockSpeed].Bins[0].Name
```

yields the bin name string "3.00GHz"¹², assuming an instance of type BinDefs named bindefs exists.

¹¹ This form requires that the program knows the in-use instance, e.g., bindefs.

¹² The returned string does not include quotes, regardless of whether they were used in the bin name or not unless *String* provides a mechanism for getting an unquoted version.

Expression:

```
bindefs.Pass.ClockSpeed[ "3.00GHz" ].counter.OnLoad
```

yields an Integer representing the number of times bin "3.00GHz" has been set since the test program was loaded.

3.2.4 Binning Syntax

Binning code is restricted to specific blocks¹³. For tests and flows, that is in the PostAction, PassAction, and FailAction blocks. For flow-nodes, that is in the PostAction and exit-port action blocks.

Statements SetBin, Stop, and SetBinStop are provided because they represent commonly used actions provided by most testers and are easily parsed by both human and machine, e.g., it is easier to find SetBinStop occurrences than the equivalent **If** statement syntax provided below.

```
bin_identifier =
<
  Bin = Bin;
  Index = Unsigned;
  Name = User_defined_name; // See 1450.0
  Number = Integer;
>

SetBin (Bin);
SetBin { bin_identifier }
SetBinStop (Bin);
SetBinStop { (bin_identifier) }

Stop;
```

Figure 7: Binning

The SetBin statement sets the bin only, i.e., does not stop. Of the two Stop statement forms, the first is unconditional, the second only stops if ContinueOnFail is False. SetBinStop semantics are equivalent to the following¹⁴:

```
If FailBin != NoBin
  { SetBin; If ContinueOnFail == False Stop; }
```

¹³ Re-evaluate with respect to Skip action, i.e., maybe PreAction binning should be allowed.

¹⁴ Lots of assumptions about syntax which is incompletely defined in syntax document.

3.2.5 NoBin Properties

NoBin is a special bin defined by the standard. It has no user-settable properties. Figure 8 shows the values returned when *NoBin* properties are interrogated. Refer to Figure 6 for the definition of *counter_reset_event*.

Color	// String "grey"
ContinueOnFail	// Boolean True
counter.counter_reset_event	// Unsigned, default = 0, incremented when set
Enabled	// Boolean True
Index	// Unsigned MaxUnsigned Infinity
IsFailBin	// Boolean False
isSet.counter_reset_event	// Boolean, default = False, becomes True when set
Name	// String "NoBin"
Number	// Integer MaxUnsigned Infinity
retest.(current Original)	// Unsigned 0 0
Terse	// String "" (empty)
Verbose	// String "" (empty)
WafermapChar	// Character ' ' (space)

Figure 8: NoBin Properties

Issues:

- Do we allow Bin assignment, i.e., Bin::operator=(const Bin&), ref Index, Number
- MaxUnsigned or Infinity are currently not keywords. What are the merits of each, assuming Infinity could be used across types, e.g., Integer, Real, Seconds, etc

3.2.6 Re-probe Syntax

Issue: define format easily mapped to multiple targets. Is there such a thing ? Delay to phase II ?

3.3 Examples

4 Spill-over Issues

Some issues arising in the context of binning spill over into other areas of the language and/or vice versa, begging for some co-ordination to minimize unnecessary proliferation of rules and concepts. These are merely alluded to here, expecting resolution in the greater context:

- a. Initialization: when defining an object such as a variable, it is useful to be able to specify the event that initializes/re-initializes it, e.g., one counter may be re-initialized by event *OnWaferStart*, another by event *OnLoad*, etc.
- b. Object member functions: it is useful to specify member functions for integral objects, e.g., `string.length()`, `array.dim()`, `array.dim(0).length()`, etc.
- c. Do we want to allow string to number-with-units conversion ?
- d. Do we provide syntax to allow the user to trigger an `EntryPoint` event, e.g., trigger `OnStart` or `OnRetest` for `retest` ? Support more standard or user-defined events, e.g., low bin yield alarms ? **Delayed until phase II.**
- e. IO syntax, e.g., `cin`, `cout`, `cerr`, `clog`, `stringstream`, file IO, formatting, etc. **Delayed until phase II.**
- f. Prior to optional property access syntax, the in-use `BinDefs` must be known (currently described in `TestProgram` block which comes last).
- g. A context sensitive keyword for the null bin is required, e.g., `NullBin` or `NoBin`. **Settled on NoBin.**
- h. Need `OnRetest` event/`EntryPoint` (initializes `retest.current` along with `OnLoad`). **Done.**

In order to adequately describe binning behavior or syntax, some clarification regarding the semantics and/or syntax associated with existing language elements is necessary. Clarification is needed for the:

- a. semantics regarding *OnStart* and *OnSiteStart* `EntryPoints`: ***OnSiteStart* has been dropped.**
- b. use of dot 0 `Spec/Category` syntax, i.e., how do we access, e.g., *Meas* under `Spec tmode_spec`, or *Max* under variable `tplh` under `Category tmode` under `Spec tmode_spec` ? Do we need a *Meas* for every *Category* (device in different state may have different measurement result for the same *Spec*) ?
- c. dot 4 provided basic test definitions: syntax requirements may change depending on what pre-defined test elements exist in dot 4, e.g., if all we provide is a functional test then in order to be define an DC or AC parametric test, syntax that permits the iterative alteration of a specific level or timing edge is required. If dot 4 provides basic tests to perform linear and binary searches that syntax may take on a different character. Run-time efficiency may be affected by our choices.
- d. definition of dimensioned array variables.
- e. Does *String* provide a mechanism for getting unquoted version ?

E. J. Wahl
ejwahl@att.net
 (304) 647-4784